Lucid: A Language for Control in the Data Plane

John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker

ABSTRACT

Programmable switch hardware makes it possible to move fine-grained control logic inside the network data plane. This allows a wide range of applications (e.g., performance-aware routing, distributed stateful access control, denial-of-service attack mitigation, and application-level caching) to operate with better performance and lower overhead than control-plane software solutions. However, these control applications are inherently hard to write in existing data-plane programming languages (e.g., P4). In this paper, we present the Lucid language, which raises the level of abstraction for putting control functionality in the data plane. Lucid supports multiple "threads" (for expressing packet-handling and control logic), syntactic constraints that ensure each stateful operation can "fit" within an arithmetic logic unit, and an ordered type system that ensures the compiler can distribute program state and logic across the hardware pipeline. Our compiler, which translates Lucid programs into P4 code optimized for the Barefoot Tofino, demonstrates the practicality of our approach. We demonstrate Lucid's generality, efficiency and ease-of-use by implementing a diverse set of ten data-plane applications with integrated control. Implementation time, even for an inexperienced developer, ranged from tens of minutes to a few hours, while resulting in 5-10x fewer lines of code compared to P4. The applications compile efficiently to real hardware, and in a case study of a stateful firewall we measure over a 200X reduction in flow setup latency.

1 INTRODUCTION

In the early days of Software-Defined Networking (SDN), controller applications changed network behavior by updating the match-action rules that switches use to forward packets. Unfortunately, many interesting applications needed the switches to direct packets to the controller (e.g., to learn about new flows and install new rules in response), causing latency, overhead, and security vulnerabilities. In addition, writing these applications was tricky, since programmers had to reason about possible inconsistencies between the switches and the controller due to delays in installing new rules. As a result, few of these dynamic controller applications saw any significant deployment in practice.

The emergence of programmable data-plane hardware has the potential to change all that, by making it possible to move fine-grained control logic into the forwarding engines of individual switches. Modern hardware, i.e., PISA pipelines [4], supports not only flexible parsing and manipulation of packets, but also updates to local state (such as register memory) that can be used to affect the handling of future traffic.

Consider, for example, a stateful firewall that protects an enterprise from unsolicited traffic. By default, packets sent by external hosts are dropped. Upon receiving outbound packets, the switch state is updated to permit return traffic from the destination; after a period of inactivity, the switch returns to dropping such packets. Having a controller handle these "events"—the arrival of the first packet and the timeout after inactivity—introduces latency and overhead, and the subtle risk that return traffic starts arriving before the data-plane is updated to permit it. Implementing this logic directly in the data plane reduces reaction time, and avoids the need for synchronization between controller and data plane.

The stateful firewall is just one of many examples. Figure 1 presents several other applications, along with their state and their data-plane and control-plane components. Past researchers have demonstrated that many of these applications, including load balancers [2, 13, 16] and caches [15], benefit substantially from data-plane implementations. Despite this, writing applications that do network control inside of PISA data planes is incredibly difficult. Languages like P4 offer the abstraction of remotely-controlled processing of a single stream of packets. However, applications like the stateful firewall require multiple "threads," one for packet handling (e.g., forwarding permitted packets and dropping the rest) and several more for control (e.g., updating the rules in response to both packet arrivals and expired timers).

Unfortunately, success in creating control applications for software switches (e.g., P4 behavioral model) is not sufficient. Indeed, P4 programs written for software switches generally do not compile to PISA switches without substantial changes, due to the constraints of hardware pipelines. First, the data plane has low-level mechanisms for updating a register's contents based on the current packet, with all sorts of nuanced constraints. Control operations can require complex actions that run up against the limits of what the Arithmetic Logic Units (ALUs) can do. Second, the data plane is a pipeline where each register array is associated with a single stage, but multiple threads of control logic may access the same shared state. To compile successfully to PISA switches, threads must access shared state in a consistent order.

Faced with these challenges, the programmer must resort to a painful trial-and-error process of rewriting the program and grappling with cryptic compiler errors, never sure when the compiler will finally yield to the next tweak of the
program. Writing a program for a single switch is difficult enough, but creating distributed control applications (e.g., for routing protocols) is even harder. These applications need inter-switch communication to detect link failures, share load statistics, and more. Ironically, the languages designed to program the network provide little in the way of abstractions to make this inter-switch communication easier! Programmers must implement “probing” from low-level mechanisms, which involves choosing a packet format, writing parsers and deparsers, configuring multicast engines outside of P4, and writing P4 code to generate and respond to the probes.

Lucid: Simple, Event-driven Data-Plane Programming. This paper introduces Lucid, a high-level programming language for implementing control applications in PISA data planes.

Lucid programs are organized as multiple collaborating components located either on a single switch or distributed across many switches in a network. In Lucid, programmers program with high-level, abstract events and handlers. Each event is named and carries user-specified data, while its associated handler defines the atomic stateful computation to perform when an event occurs. An event could be a packet to process, a request to install an entry in a firewall, or a probe from a neighboring switch to report a link’s status.

Events are also associated with times and locations to facilitate coordination between control operations among different switches, possibly with a delay. Programmers can write sophisticated control logic without having to worry about the low-level details of custom packet formats, parsers and deparsers, or having to “roll their own” mechanisms for buffering and delayed information processing.

Lucid’s event-based abstractions for structuring applications and coordinating control are complemented by a careful “correct-by-construction” design of the core language. Rather than allowing programmers to write arbitrary event handler code which may fail to compile to a PISA pipeline, Lucid uses several techniques in concert to avoid, or detect and respond to, programmer errors at the source language level:

- **Syntactic constraints**: We define domain-specific, source-language constraints to characterize memops, stateful operations that can execute in a single ALU of a PISA switch. Memop definitions that cannot fit in a single ALU are rejected, and source-level error messages point out exactly where any such mistakes occur, making it easier for programmers to understand how and why they must change the processing of an individual control operation.

- **Types and effects**: We develop a novel ordered type-and-effect system that limits the way programs interact with persistent memory. Our ordered type system tracks the order in which handlers access registers and provides actionable source-level feedback when there are inconsistencies. This helps programmers quickly identify control operations that must be adapted (e.g., decomposed into multiple simpler events).

- **Well-designed primitive abstractions**: We develop line-rate persistent data structures as a first-class language feature, e.g., arrays that can be passed as arguments to functions. These abstractions provide a powerful high-level interface to state that exploits our domain-specific type system and syntactic constraints.

Collectively, these Lucid abstractions give programmers a natural and modular way to express data-plane applications that interleave packet processing with ongoing control operations, and help them navigate the difficulty of programming complex switch hardware at a high level of abstraction.
The Lucid compiler shows how to map the above ideas to real hardware—the Barefoot Tofino. Our compiler analyzes Lucid programs and translates valid programs into a lower-level intermediate language for optimization by exploiting instruction-level parallelism. The compiler generates a Tofino-compatible P4 program and links it with a lightweight event scheduler that runs in the data plane.

We evaluate Lucid by using it to build a diverse set of data-plane programs with integrated network control and compile them to the Tofino. Applications include a stateful firewall, fault tolerant router, self-driving DNS protection service, telemetry cache, and more. All programs had 5-10X fewer lines of code than their P4 equivalents and, due to the Lucid compiler’s optimizations, utilize the Tofino’s limited pipeline stages efficiently.

In an informal user study, we also find that Lucid significantly lowers the barrier of entry to high-speed data-plane programming. Programmers who have never worked with the Tofino before are able to implement and compile non-trivial applications (i.e., with stateful operations and multiple threads of control) in tens of minutes.

Finally, a case study of a stateful firewall demonstrates the performance benefit of using Lucid to integrate latency-sensitive network control in the data plane. We measure over a 200X reduction in flow setup latency for the stateful firewall.

In summary, the contributions of this paper are:

- Lucid, a high-level language for data-plane programs with integrated control routines that feature complex stateful and distributed operations;
- a novel type system, syntactic restrictions, and static analysis passes that identify Lucid programs ill-suited for the underlying hardware and provide actionable source-level feedback to the programmer;
- a compiler targeting the Barefoot Tofino, showing how Lucid’s high-level event-based abstractions can be implemented and optimized in practice; and
- an evaluation of Lucid demonstrating that it has broad applicability, reduces programmer effort, and enables high-performance control in the data plane.

Ethics: This work does not raise ethical concerns.

2 INTEGRATED DATA-PLANE CONTROL

This section introduces our core mechanisms for enabling control operations in PISA switches. We explain these mechanisms through a driving example: the fast rerouter, a forwarder with fault detection and routing in the data plane. We are motivated to push network control into the data plane to avoid the inherent latency of solutions where detection and routing are done remotely [12], so that our system can react at (or near) the speed of the underlying hardware [13, 16].

The fast rerouter has three components:

- Forwarding. The fast rerouter looks up a next hop for each packet in an associative array based on its destination address. Before sending the packet out, the node checks a second data structure to determine if the next hop is still reachable. If not, it increments a failure counter.
- Fault detection. Concurrent with forwarding, a node also regularly pings all of its directly connected neighbors to determine if they are still reachable.
- Routing. Interleaved with the above, each node also periodically scans its data structures for routes with unreachable next hops. When one is found, the node queries all of its neighbors to find the shortest alternate route.

2.1 PISA Programmable Packet Processing

The target for our fast rerouter is a PISA switch. The core of a PISA switch, illustrated in Figure 3, is a programmable line-rate match-action pipeline. Line rate demands a tightly synchronized feed-forward design: each pipeline stage has a throughput of one packet per cycle and packets only ever move forward through the pipeline. Instruction-level parallelism is also critical for line rate. A packet’s header moves through each stage in parallel, as a vector. When the packet header enters a stage, ternary (TCAM) and exact (hash + SRAM) match-action tables evaluate it to feed ALU vectors with instructions to modify header fields. The “programmable” aspect of the pipeline is the capability to set table layouts and instructions at compile time, and set table entries from a management CPU at run time.

Stages also have stateful ALUs (sALUs) for updating local SRAM register arrays. Each stateful ALU can read from a single address in SRAM, perform limited computation, and write back to SRAM or modify metadata associated with the packet. All memory operations done in a pass through the pipeline are atomic [23] because of the feed-forward design.

After the match-action ingress pipeline, a packet can be directed to one or more egress pipelines for further processing before transmission. Alternatively, the packet may be sent through a dedicated recirculation port that brings it back to the start of the pipeline for additional processing. Finally, there are semi-programmable “support engines” outside of the...
pipeline: a multicast engine to copy packets, a queue manager to isolate and shape flows, a packet generator for spawning packets, and configurable MAC blocks that can dynamically pause queues based on Priority Flow Control (PFC) frames.

2.2 Atomic Control Operations via Packets

Clearly, PISA pipelines are highly optimized for packet processing. How can we use them for control applications with shared state, multiple threads, and distributed computation?

The key observation is that control-plane tasks can be broken down into atomic operations that map to individual packets. For example, we have already described the fast rerouter’s routing component as three atomic operations: checking the status of a single route, answering a single query from a neighbor, and processing a query response. Each of these operations can be performed in a single pass through a PISA pipeline, compiled to a “slice” of its parallel ALUs, sALUs, and match-action tables, as illustrated in Figure 3.

Once we map individual control operations to packets, we can interleave them with regular data packets, as shown in Figure 3. This approach to concurrency gives us a powerful and intuitive memory model: each operation has atomic access to the data-plane state using stateful ALUs.

2.3 Fast Control Threads via Recirculation

To get from simple control operations that happen instantly to complex control operations that take place over time, e.g., from an operation that scans one route entry to a thread that repeatedly scans an entire route table, we can use packet recirculation. We can use recirculation for serial processing, by recirculating a control packet multiple times to perform one part of its task in each pass, or we can use it for parallel processing, by recirculating multiple control packets back-to-back, each operating on a different chunk of data.

In either case, using even a small fraction of the pipeline bandwidth gives us blisteringly fast control. For example, in the fast rerouter, suppose we use one of the Tofino’s dedicated 100 Gb/s recirculation ports for control operations. That is approximately 100M control operations per second. Assuming each control operation checks a single route, the fast rerouter could identify all the failed routes in a table with 65K entries in under 1 ms. This is less than one tenth of the time that it takes a optimized remote control thread [27] to simply read the table from the pipeline. Further, this leaves the switch with bandwidth for millions of other control operations and billions of regular data packets per second.

2.4 Scheduled Control via Support Engines

Of course, the control application may not always want control threads to operate at such high rates, or, for that matter, at the same switch. For example, the fast rerouter needs to periodically ping its neighbors to check if they are still reachable. This brings us to the last piece of the puzzle: what components of a PISA switch can we use to schedule the place and time where control operations execute?

Place. Changing where an operation executes is straightforward, assuming that switches have addresses. Since control operations are processed like packets, a switch can schedule an operation at another location by encapsulating the corresponding control packet in an appropriately addressed frame and forwarding it just like any other packet. With line-rate multicast engines, we can even schedule an operation at multiple locations (such as the fast rerouter pinging all neighboring switches) in a single step.

Time. Changing when an operation executes is harder. Essentially, we need to buffer a control operation for some amount of time. A simple design would buffer it in a register array along with the time at which it should be executed, and then scan the array periodically to find operations ready to execute. However, this approach could consume a large number of stateful ALUs. A simple alternative is to just recirculate a control packet repeatedly until it is ready to execute, but this consumes recirculation bandwidth. A more efficient mechanism is to use a dedicated queue for delayed control operations, which is paused and periodically released using PFC pause frames from the switch’s own packet generator.

3 EVENT-DRIVEN PROGRAMMING

To create control applications for PISA switches, programmers currently must implement many low-level mechanisms by hand. In many ways, it is reminiscent of writing a distributed system without basic operating system services. Consider the challenge of adding the fast rerouter’s route query control operation to a basic forwarding program written in P4. We must define a route query header along with parsers and deparsers; adjust the control flow to branch on that header’s presence (in addition to existing branches); serialize generated queries into event packets; and finally configure the multicast engine to broadcast these packets to all neighbors. All of this effort only gets us to the point where we can begin to implement the interesting logic, e.g., the P4 blocks that generate and respond to route queries.
3.1 Event-based Lucid Abstractions

The main idea behind Lucid’s core abstractions is to unify packet processing with control operations through intuitive primitives for coordinating when and where events execute.

Events. Lucid abstracts both control operations and data packets as events. Every packet carries at most one event, which consists of a four-tuple containing (1) a name, (2) carried data, (3) a time, and (4) a place. Events give programmers a way to structure multi-threaded programs that is missing from P4 and other existing data-plane languages. For example, the routing component of the fast rerouter has three events, corresponding to its three operations in Figure 2.

```
event route_query(int switch, int dst);
event route_reply(int switch, int dst, int pathlen);
event check_route(int dst);
```

Events are also a high-level abstraction for application-layer messages. The route_query event is a request that switch sends to its neighbor, asking for the length of its path to dst. A route_reply is a response to a query. Finally, check_route is an instruction that a switch sends to itself to check whether the route to dst has failed.

Handlers. A handler specifies what happens, such as a control operation or a packet-processing function, when a switch gets an event. Handlers are atomic units of computation in a Lucid program. Each handler compiles to a slice of parallel tables, ALUs, and stateful ALUs, and executes in a single pass through the match-action pipeline. Although the low-level implementation of a handler is complex, the language for writing the handler is simple and expressive. For example, here is the route_query handler from the fast rerouter:

```
global pathlens = new Array<32>(){tbl_x2};
memop incr(int stored, int i) { return stored + i; }

handle route_query(int neighbor_id, int dst) {
    int pathlen = Array.get(pathlens, dst, incr, 1);
    event reply = route_reply(SELF, dst, pathlen);
    generate Event.locate(reply, neighbor_id);
}
```

The handler runs on a switch when its neighbor neighbor_id schedules a route_query event to execute on it. The handler looks up the length of the path to dst from a persistent array (pathlens), increments the value by 1, and returns the result to neighbor_id by scheduling a route_reply event to execute there. The programmer can implement all the logic for a route query while only writing roughly the number of lines it would take merely to declare a route query header in P4.

Stateful abstractions. Lucid is so concise because of its well-designed primitive abstractions for stateful computation, which is at the core of network control logic. This example shows two of the key abstractions.

- First, persistent line-rate data structures like pathlens, a persistent array that stores 32-bit integers. These data structures provide a high-level abstraction over the stateful computation that can be done in a single PISA pipeline stage.
- Second, Lucid also introduces a flavor of functional programming in PISA switches, which improves modularity and code reuse. The route_query handlers passes Array.get a function, incr, to use when calculating a return value. incr is a memop, a special type of function that is allowed to operate on persistent memory. It can be re-used across the program every time a handler needs to increment a value from some array. Programmers can also write functions called directly from handlers, taking either local variables (e.g., pathlen, which only persists for the handler’s execution), or persistent data structures as arguments.

These abstractions are not just shallow wrappers of P4. They are realized with a sound type system that tracks where and how a program uses state. As Sections 4 and 5 will discuss, this frees developers from some of the most challenging aspects of programming a PISA pipeline, making it easy to build complex applications that combine state and computation in novel ways.

Event generation. As the route_query example also shows, handlers not only perform some computation in response to an event/packet, but can also generate events to trigger additional future computation. This simple idea is extremely powerful. It lets Lucid express a much broader class of computation than current data-plane languages for PISA switches, by providing an abstraction for time.

This is a critical feature for integrated control in the data plane, because it lets us break up complex operations that can (or must) happen over a period of time. For example, in the fast rerouter, we implement the thread that periodically scans the status of every route request as a recursive event. The event is essentially an instruction to check the status of a route at a certain position in the routing table, then generate another event to check the next position. As another example, the stateful firewall application (Section 6.3) uses an event that recurses a bounded number of times to implement the insert operation of a cuckoo hash table in the data plane.

Event combinators. Event combinators are high-level primitives that let a handler change when and where an event that it generates will execute.

The locate combinator, which the route_query example uses, lets a handler specify where an event will be generated. We can think of this as the equivalent of a send system call in Linux. It provides a simple abstraction for distributed elements of a program to communicate. Lucid provides combinators for unicast and multicast event generation.

The delay combinator changes when an event is executed. This makes it easy to pause persistent computations, similar to the sleep system call in Linux. The fast rerouter uses the
3.2 Data-Plane Event Scheduler

Lucid realizes the abstractions for event-based distribution and communication using an event scheduling library that is
inlaid into a Lucid program. As Figure 4 shows, the library s
logically between a Lucid application and the underlying
network, filling the role of a lightweight operating system.

We describe the main components by following the event
trace in Figure 4, beginning with the execution of handler a
in the ingress pipeline of switch 1. This generates two events,
b and c, by removing the a event header from the packet and
attaching event headers for both b and c.

Event serialization. After the ingress pipeline finishes, Lu-
cid’s serializer transforms the single packet with headers for
b and c into serialized event packets, one for each event.

First, the serializer uses the switch’s multicast engine to
create one copy of the packet for each event. When a copy
arrives at the egress pipeline, it has headers for both b and
c. The event serializer deletes one header from each copy,
using a clone ID field that the Tofino provides as metadata.

Event dispatching. The event serializer sends the event
packets for b and c to the switch’s recirculation port. When
these packets re-enter the ingress pipeline, the event data is
extracted by a Lucid-generated parser and passed to an event
dispatcher, an ingress match-action table that performs one
of three actions based on an event’s location and delay.

Non-local events: If the event’s location is not the current
switch, the dispatcher calls a user-configured forwarding
table to select an output port or multicast group for the
event. In the example program, the dispatcher at switch 1
sets a multicast group for the event packet c, which will send
copies of the packet to switches 2 and 3.

delay combinator to control the rate at which it pings its
neighbors and also the rate at which it scans its routing table.

Delayed local events: For events that are destined to the
local switch, but with a delay > 0, the dispatcher calls a delay
function. In the example, switch 2 initially delays event c.

Processable events: When an event’s delay is 0 and its loca-
tion is the current switch, the dispatcher applies a sequence
of Lucid-generated tables that implement the event handlers.
In the example program, the dispatcher at switch 1 will do
this for b as soon as it arrives.

Implementing delay. Delay is the most sophisticated func-
tion in the scheduler. Lucid implements this with pausable
egress queues. Events to delay are placed in a special “delay
queue” of the recirculation port. The queue is paused most of
the time and unpaused at a regular interval to release packets,
e.g., once every 100 µs. When events exit the queue, a table
egress updates their delay parameter based on their queue
time. The packets recirculate and repeat until their delay is 0.
PFC (Priority Flow Control) packets let the event scheduler
time the queue. It configures the switch’s packet generator
to produce pairs of PFC packets that arrive at a low, constant
rate. The first PFC packet in a pair unpauses the queue to let
event packets out, while the second one repauses it.

4 PISA PIPELINE CONSTRAINTS

The logic of P4 programs constrains the layout of stateful
data along a PISA pipeline in a variety of ways. A P4 compiler
tries to solve these constraints and allocate stateful data to
particular stages of the pipeline. Unfortunately, it may fail,
and when it does, programmers are left with little guidance
as to how to fix their programs. Lucid avoids some of these
issues by interpreting a program’s data declarations as an
implicit, high-level specification of the programmer’s data
layout intentions. The Lucid type system then verifies that
the order of data accesses in the rest of the program is con-
sistent with the specification, guaranteeing that compilation
is possible (if enough pipeline stages are available). When
an access ordering error arises, a useful source-level error
message indicates the specific lines of code in conflict.

4.1 Ordered Data Access

Limitations of the Tofino hardware mean that any persist-
tent, mutable data (such as arrays or counters) must be stored
in registers within the packet-processing pipeline. These reg-
isters are partitioned across the stages of the pipeline, and
hence the data must be partitioned as well. This leads to a
natural order in which each handler must access the data.

To illustrate this issue, Figure 5 presents a simple but
invalid Lucid program. The program declares two arrays
arr1 and arr2, and two handlers setBoth and setCond which
access those arrays in different orders. In general, programs
of this form cannot be compiled to the Tofino—one handler

```c
event a(); event b(); event c(); const group GRP = (2, 3);
handle a() {
  generate b();
  mgenerate Event.delay (Event.locate (c(), GRP), 10ms);
}
```

Figure 4: Event scheduling for a simple program.
write, as they must declare and initialize their data anyway. The specification is also quite high level as it does not refer to encapsulate common idioms and user-defined abstractions. 

PISA programmers must typically ensure that their code fits within specific hardware stages; in fact, programs such as this are designed for enforcing protocols such as open-read/write-close sequences over OS resources are unnecessarily complex, requiring additional type annotations or sophisticated inference mechanisms. In Lucid, the key difference is that we know all the ordered variables in advance, allowing us to create a simpler system that can still define and verify functions separately from where they are used.

At a high level, our strategy is to use a system in which effects are integers representing abstract stages. Each ordered variable (either an array or a counter) is associated with a stage based on the order in which variables are declared. During type checking, we keep track of a current stage, which tracks the most recently-accessed ordered variable. The type checking fails if the program attempts to access an ordered variable whose stage is less than the current one, indicating that during execution the packet would have already passed by that data in the pipeline.

4.2 Types for Ordered Data Access

We use a type-and-effect system to check that a Lucid program is well-ordered while also performing regular type checking. While past work [5, 14] explored the use of ordering constraints to ensure correct access to volatile state in other contexts (e.g., “no-use-after-free” properties for memory managers or “no data access without first acquiring a lock”), we are unaware of prior uses of ordered type systems for data layout along pipelines, or more generally in the context of networking. On the one hand, systems such as ordered logic [21] appear too restrictive for our purposes—functions that refer to an ordered variable cannot be declared until prior variables are used. On the other hand, prior systems [5, 14] designed for enforcing protocols such as open-read/write-close sequences over OS resources are unnecessarily complex, requiring additional type annotations or sophisticated inference mechanisms.

We say a program is well-ordered if the data accesses in every handler follow the same order as the global data declarations. In other words, we treat the order of data declarations as a specification that clearly documents requirements for all handlers. It is an easy specification for programmers to write, as they must declare and initialize their data anyway. The specification is also quite high level as it does not refer to specific hardware stages; in fact, programs such as this are portable across different hardware platforms with different numbers of stages. The compiler has the flexibility to place any object in any stage so long as it faithfully implements the program’s semantics. The specification merely ensures that, provided a program adheres to its requirements, the compiler can find some solution to the data-allocation problem.

If programmers do not adhere to the specification, a simple error message directs them to the disordered portion of their program. For the program in Figure 5, Lucid would issue an error for the setCond handler saying that it accesses arr2 and arr1 in the opposite order of their declarations. Ordering errors are detected by a type-and-effect system described in Section 4.2. While verifying a toy example like Figure 5 is straightforward, the verification problem becomes increasingly difficult as programs grow and use auxiliary functions to encapsulate common idioms and user-defined abstractions. Such functions may access global variables (directly or via arguments), which constrains the order in which they may be called from other functions or handlers.

Figure 5: A disordered program

```c
const int SIZE = 16;

global arr1 = new Array<32>(SIZE);
global arr2 = new Array<32>(SIZE);

handle setCond(int idx, int data) {
    int x = Array.get(arr1, idx);
    if (data > x) {
        Array.set(arr2, idx, data);
    }
}

handle setBoth(int idx, int data1, int data2) {
    Array.set(arr2, idx, data1);
    Array.set(arr1, idx, data2);
}
```

We use a type-and-effect system to check that a Lucid program is well-ordered while also performing regular type checking. While past work [5, 14] explored the use of ordering constraints to ensure correct access to volatile state in other contexts (e.g., “no-use-after-free” properties for memory managers or “no data access without first acquiring a lock”), we are unaware of prior uses of ordered type systems for data layout along pipelines, or more generally in the context of networking. On the one hand, systems such as ordered logic [21] appear too restrictive for our purposes—functions that refer to an ordered variable cannot be declared until prior variables are used. On the other hand, prior systems [5, 14] designed for enforcing protocols such as open-read/write-close sequences over OS resources are unnecessarily complex, requiring additional type annotations or sophisticated inference mechanisms. In Lucid, the key difference is that we know all the ordered variables in advance, allowing us to create a simpler system that can still define and verify functions separately from where they are used.

At a high level, our strategy is to use a system in which effects are integers representing abstract stages. Each ordered variable (either an array or a counter) is associated with a stage based on the order in which variables are declared. During type checking, we keep track of a current stage, which tracks the most recently-accessed ordered variable. The type checking fails if the program attempts to access an ordered variable whose stage is less than the current one, indicating that during execution the packet would have already passed by that data in the pipeline.

Formally, our typing judgement has the form \( \Gamma, e_1 \vdash e : \tau, e_2 \). In English, this can be interpreted as the statement “Starting with environment \( \Gamma \) and at stage \( e_1 \), the expression \( e \) has type \( \tau \) and will finish evaluating in stage \( e_2 \).” We use this to prove the following soundness theorem:

**Theorem:** If \( \emptyset, e_1 \vdash e : \tau, e_2 \), then either \( e \) is a value, or \( e \rightarrow e' \) and there is some \( e'_1 \) such that \( \emptyset, e'_1 \vdash e' : \tau, e_2 \).

This theorem implies that any program which typechecks will never “get stuck” trying to access unavailable data. The full details of the type system, and the proof of the theorem, appear in Appendix A.

5 PISA ALU CONSTRAINTS

PISA programmers must typically ensure that their code fits the capabilities of the individual ALUs of the pipeline. This is difficult in simple packet processing, and quickly becomes daunting for more complex applications that integrate control processes. This section describes the two mechanisms that Lucid introduces to free developers from this burden.

---

1In this case, a compiler could reorder the two statements of setBoth (as the latter statement does not depend on data from the former), placing them in the same order as setCond. However, to simplify the rules of the programming model for the programmer, our compiler does not perform such reorderings. Such reorderings are not possible in general.
5.1 Constraining Stateful Operations

The primary goal of most control tasks, regardless of where they run, is to update and maintain the state used to handle data packets. This makes stateful operations critical in data-plane applications with integrated control.

However, stateful operations are challenging to write. The core problem is not necessarily that the underlying stateful ALUs have limitations, but rather that their nuanced constraints are left implicit in today’s data-plane languages. For example, in P4, programmers write RegisterActions with arbitrary blocks of C-like code to process a single register access. Unsurprisingly, it is easy to write a RegisterAction that cannot compile, but difficult to figure out why. The “decision” that a particular register action is too complex is made in part of a compiler’s back-end that is far removed from the source code. When compilation fails, programmers have no direct way of figuring out what went wrong. For example, here is a register action that does not compile:

```c
void apply(inout bit<32> memCell, out bit<32> retCell) {
    if (memCell > hdr.ip.src_addr) { retCell=memCell; }
    else if (memCell <= hdr.ip.src_addr) { tmp=hdr.ip.src_addr; }
    else {tmp=memCell;}
    retCell = tmp;
}
```

After some trial and error (including the message error: RegisterAction too complex), we can finally get this to compile by rewriting it as:

```c
void apply(inout bit<32> memCell, out bit<32> retCell) {
    bit<32> tmp;
    if (memCell > hdr.ip.src_addr) { retCell=memCell; }
    else if (memCell <= hdr.ip.src_addr) { tmp=hdr.ip.src_addr; }
    else { tmp=memCell; }
    retCell = tmp;
}
```

MemOps. Lucid avoids scenarios like the one above by making restrictions on stateful operations explicit and obvious to the programmer. We do this by providing a syntactic description of what these operations must look like. We call these memops, and each is a function of two arguments whose body consists of either a single return statement or a single if statement containing one return statement in each half. Furthermore, each variable may be used at most once in each expression, and only certain binary operators are allowed. When a user declares a memop, we automatically check that its body satisfies these requirements. If this check passes, the programmer is guaranteed that the operation will compile; if not, our compiler can explain exactly what is wrong.

For example, the stateful operation from above would be:

```c
memop get_max(int stored, int addr) {
    if (stored > addr) (return stored; }
    else { return addr; }
}
```

A memop is then used by passing it to Array method calls throughout the program, as shown above for get_max. The Lucid compiler translates each call that uses a memop into a P4 register action that compiles to the Tofino. The restricted syntax of memops means that they have only a finite number of possible forms, making it easy to verify that all of them do in fact compile.

5.2 Simplifying Stateless Operations

Operations over non-persistent variables (i.e., variables stored in packet headers or metadata) also face constraints based on the capabilities of individual ALUs. In practice, programmers can often avoid issues with careful programming discipline. For example, on the Tofino:

1. Break down each operation over packet metadata into a sequence of simple operations, where a “simple operation” is an expression that maps directly to a single Tofino ALU instruction.
2. Encapsulate each “simple operation” in its own action and table, to ensure that no passes in the P4 compiler fold multiple simple operations together.

This will produce a program that does not violate ALU-level constraints, but is extremely tedious for programmers. Lucid’s compiler eliminates this burden by automatically translating every statement in a Lucid handler into a sequence of single-action P4 tables. Every single-action P4 table is simple enough to use at most one ALU or stateful ALU, and thus can always compile to a PISA switch. There are three main types of single-action tables.

**Operation table:** An operation table uses a single ALU to evaluate a binary expression over two local variables and assign its result to a third local variable. For example,

```c
void apply(inout bit<32> mem, out bit<32> ret) {
    ret = (mem > hdr.ip.src_addr) && (mem <= hdr.ip.dest_addr);
}
```

**Memory operation table:** A memory operation table uses a single stateful ALU to update one element in a P4 register array. It is a direct translation of a call in Lucid to an Array method that is passed a memop. For example,

```c
Array.setm(tcp_cts, port, plus, 1); translates to:
RegisterAction<bit<32>,bit<32>,bit<32>>
{ tcp_cts } setm_1 = {
    void apply(inout bit<32> mem, out bit<32> ret) {
        ret = (mem = mem + 1);
    }
    action do_setm_1() { setm_1.execute(port); }
    table tbl_setm_1 { actions = {do_setm_1; }
    const default_action = {do_setm_1; }
}
```

**Branch table:** A branch table uses the table itself to compare a local variable to a constant. It calls either a true or false action. These actions determine which execution path
a program takes in the subsequent stages of a PISA pipeline. For example, if `if (proto != TCP)` translates to:

```pseudocode
action if_true(); action if_false();
table tbl_if {
  keys = (proto : ternary);
  actions = {(if_true; if_false;)
    entries = {
      (TCP) : if_false;
      (...) : if_true;
    }
}
```

Since each single-action table maps directly to a primitive in the PISA pipeline, a Lucid program represented as a sequence of such tables will never violate ALU-level constraints.

### 5.3 Minimizing Number of Pipeline Stages

By transforming a Lucid program into a sequence of single-action tables, we produce P4 code that may span many pipeline stages. To address this issue, the Lucid compiler applies two optimizations that greatly reduce the number of stages required by the resulting P4 code.

**Inlining branch operations.** Branching on a table is expensive in a PISA switch because the tables that come next in the program must be placed in a subsequent stage. For example, in the following P4 code, tbl_B and tbl_C must be placed in a stage after tbl_A.

```pseudocode
switch(tbl_A.action_run()) {
  if_true : {tbl_B.apply(); }
  if_false : {tbl_C.apply(); }
}
```

This makes the single-action table representation of a Lucid handler require many stages because of its branch tables. For example, Figure 6(1) shows this representation of a simple Lucid handler. It requires seven pipeline stages to execute, three of which are used by branch tables.

The Lucid compiler eliminates the need for branch tables by transforming the operation tables (i.e., the tables that do not branch) so that each table checks all the conditions necessary for its own execution, using static match-action rules. If the conditions are met, the table executes its single action. If the conditions are not met, the table executes a no-op. For example, consider the table `idx_eq_0`. By following the control path from the root node to the table in Figure 6(1), we see that it only executes when `proto != TCP` and `proto != UDP`. Thus, in Figure 6(2), the `idx_eq_0` table only executes its action when these conditions are true.

**Merging tables and actions.** The Lucid compiler rearranges tables to further reduce the number of stages used by a program. For example, in Figure 6(2), the table that implements `Array.set(hcts, dst, plus, 1);` does not actually need to execute last. It does not have any data flow dependencies on previous tables. That is, its execution condition and input variables are not modified by any other tables in the program, so it can be executed in parallel with the first table, as shown in Figure 6(3).

![Figure 6: Optimizations of the P4 representation of a Lucid handler. The single-action table representation (on the left) requires seven pipeline stages. By inlining conditional tests to eliminate tables dedicated to branching (top right) and then reordering tables according to their data dependencies (bottom right), the Lucid compiler reduces this to four pipeline stages.](image-url)

Ultimately, the only factors that matter for determining the layout of tables are data flow dependencies and the resources (tables, ALUs, etc) in each stage. Tables themselves are a primary bottleneck—there are typically far more ALUs than tables in a PISA stage.

This motivates Lucid’s second main optimization: it merges all the single-action tables into a layout of multiple-operation tables according to data dependencies. This is possible because Lucid-generated tables use only static rules. Figure 7 shows how tables from the example in Figure 6 get merged.

The algorithm is a simple greedy instruction selection. It takes a list of single operation tables to place and two parameters, M and N, that define the number of stages in...
We first compare the lines of code required to program in Lucid versus P4. *Flow [26] is a complex application that gives us a point of comparison to hand-written P4. The Lucid program is a complete implementation of *Flow in 149 lines of code. The published implementation, in P4_14, is 1559 lines of code—over 10X longer.

We are unaware of hand-written implementations for the other Lucid applications and, due to the time required to program the Tofino in P4, we did not re-implement any ourselves. However, using *Flow as a calibration point suggests that the Lucid compiler produces P4 that is within 15% the length of hand-written P4. Thus, we conclude that Lucid reduces lines of code by around 10X for diverse applications.

Figure 9 breaks down the lines of P4. Actions and tables take the most lines. An interesting observation is that for most applications, the entire Lucid program was fewer lines of code than just the register actions in P4. This is partially because P4 register actions are not reusable like Lucid’s memops—the programmer must manually copy the code.
every time they want to repeat the same operation on a different array. In Lucid programs, and even across programs, we re-use the same generic memops multiple times.

We did a user study, measuring how long it took a graduate student who has never programmed the Tofino before to implement four applications in Lucid and compile them to the Tofino. As Figure 10 shows, all applications took around 30 minutes to an hour to implement. Achieving a compiling Tofino prototype of a new, non-trivial application in under an hour with P4 is difficult to imagine, even for an experienced Tofino programmer. For students new to the architecture, we typically find that it can take weeks to do something non-trivial. Needless to say, we are excited by the potential for Lucid to save future students’ time.

### 6.2 Optimization Benchmarks

**Compiler optimizations.** Next, we evaluate Lucid’s compiler optimizations by comparing the number of required stages with and without optimizations. Figure 11 shows the ratio for each application. For unoptimized stages, we report the number of P4 tables in the longest code path, as many programs did not fit into the Tofino’s pipeline without optimization. For most applications, optimizations reduced stage requirements by a factor of 1.5-4. The benefit was greater for complex applications, such as "Flow and the closed-loop DNS defense system, which originally had critical paths nearly 4X too long for the Tofino’s pipeline.

Figure 12 measures the number of Lucid statements that the compiler mapped to each stage with optimizations enabled. It ranged from 2 - 13, which is significant for programs that are only 100 lines of code. This tells us that the compiler is efficient, and also that many types of control processing can be parallelized effectively for data-plane architectures, which is itself a somewhat interesting result.

**Event scheduler optimizations.** A key optimization in Lucid’s event scheduler is the pausable queue mechanisms for reducing the overhead of delaying events via recirculation. We measured the bandwidth overhead and timing accuracy of delaying 64B event packets on one of the Tofino’s 100 Gb/s recirculation ports, with and without the pausable queues.

As Figure 13 shows, the pausable queues make the recirculation bandwidth cost of delayed events negligible. The bandwidth cost for delaying 90 concurrent events indefinitely was 5.5 Gb/s. In comparison, delaying 90 concurrent events without Lucid’s pausable queues consumed over 95 Gb/s—the port was effectively saturated.

This nearly 20X reduction in overhead came at only a slight increase of timing variance. As Figure 13 shows, event delay was off by up to approximately 50 µs when using pausable queues. This is because the queues are only released intermittently—every 100 µs in these trials. Users can reduce the release timing parameter to improve accuracy, at the cost of requiring a higher rate stream of pause frames.

### 6.3 Application Performance

Finally, to showcase the performance improvement that applications can achieve by integrating control into the data plane with Lucid, we benchmark the stateful firewall and compare with a remotely controlled baseline implementation.

The primary metric is flow installation time. This is critical
because a stateful firewall must install a new return entry in between the time when the first packets of an authorized flow and its return flow arrive. If not, the firewall will disrupt flows by either dropping return packets or buffering the first packet of every new flow until the installation is complete.

**Implementations.** We compare a Lucid stateful firewall with flow installations from the data plane and a Mantis [27] implementation with flow installations from the switch CPU.

The Lucid program features a stash-based Cuckoo hash table [17]. When a packet from a new flow $f$ arrives, it attempts to install $f$. If possible slots for $f$ are taken, the handler generates a flow installation event to free one of $f$’s slots. This causes up to $O(\log n)$ recirculated events, with each event performing two “Cuckoo” (i.e., key move) operations.

The comparison implementation is highly optimized. Mantis itself has orders of magnitude less latency than traditional SDN controllers [27]. The install thread runs in Mantis’s optimized Tofino driver, using a full switch CPU core to poll a register array where the data plane puts new flow keys. For each new flow, it installs a P4 exact-match table entry.

**Installation time.** We measure installation time for flows that arrive in groups of varying sizes. Figure 14 shows that Lucid’s flow installation time is **200X better for flows that arrive individually** and **1000X better for flows in groups of 50**.

Average flow installation time for Lucid was only 80 ns. Most flows (over 90%) installed at line rate, in the first pass through the pipeline. 70% of the remaining flows installed in a single recirculation—about 600 ns. The worst case was 79 recirculations—around 48 µs. This is far below most network’s round trip times, so would not disrupt traffic.

In comparison, average installation time for the remotely-controlled firewall was around 20 µs when flows arrive one at a time. When flows arrive in groups, however, queuing delays had a multiplicative effect that made flow installation times prohibitively high. For example, when the group size was 50, the average flow installation time was 496 ms.

## 7 RELATED WORK

**Control-plane languages.** Over the past decade, a variety of new control-plane programming languages have been proposed. Languages like Flowlog [19], Frenetic [6], NetKAT [3], and McNetKAT [24] enable richly programmable remote controllers. Such languages enabled many dynamic control applications [1, 10, 22], but performance was a significant issue due to the inherent overhead of remote control. Lucid resolves this by providing an expressive language for high-performance dynamic control in the data plane.

**Data-plane languages.** More recently, researchers have begun to design higher-level imperative languages for the data plane, including Chipmunk [8], Domino [23], Lyra [7] and P4All [11]. These systems lift the level of abstraction for packet processing. Unlike Lucid, they do not provide abstractions for event-driven programming paradigm. A further distinction is that these languages focus on program synthesis and optimization technology, rather than source-level program validity checking. When programmers write programs that can be compiled to the underlying hardware, the technologies work well. However, when programmers write a program that cannot be compiled (something that happens regularly even to experts), these languages have nothing to say. Typically, compilation is attempted with little feedback about what goes wrong when it fails. In contrast, Lucid defines source-language constraints to deliver a “correct-by-construction” model that avoids the problem of having “valid” source-level programs that fail to compile. It seems likely that Lucid’s correct-by-construction design and abstractions for integrated control could be added to some of these other systems. Conversely, Lucid’s back-end compiler may benefit from some of the optimization technology developed elsewhere. In this sense, these research efforts are largely orthogonal and complimentary to one another.

**Reducing control-plane overhead.** Several systems [9, 25, 27] propose moving control-plane routines from a remote server to a switch management CPU. These systems improve performance, but are still subject to the inherent bottlenecks of off-chip communication (e.g., PCIe latency [20]). Lucid follows this line of work to its ultimate conclusion, by pushing the control into the data-plane hardware itself.

## 8 CONCLUSION

Lucid makes it easy to write data-plane applications with high-performance integrated control. PISA switches already have all the necessary mechanisms; however, programmers today must use them at a very low level. Instead of writing packet-processing functions that always execute here and now, Lucid programmers can use intuitive event-based abstractions to distribute control in both time and space.

Complementing this is a careful correct-by-construction design of the core Lucid language through syntactic constraints, a sound type system, and well-designed abstractions for persistent state. We realize these ideas for the Barefoot Tofino, with an optimizing compiler that generates efficient, Tofino-compatible P4. We implemented a diverse range of Lucid applications. These applications require ~10X fewer lines of code, compared to P4 equivalents. Programmers without any prior Tofino experience were able to write compiling code in tens of minutes. Finally, a Lucid stateful firewall outperformed a remotely-controlled baseline by over 200X.

Overall, Lucid is general, fast, and easy to use. It will save time, enable new applications, and perhaps change the way we think about what hardware data planes can do.
Lucid: A Language for Control in the Data Plane

Submitted for review to SIGCOMM, 2021

REFERENCES


This appendix presents a simplified definition of Lucid’s type system, as well as an operational semantics and soundness proof. To begin, figure 15 defines a grammar for a model ML-like language on which we will define our type-and-effect system. We present the system for this language purely for convenience; adapting the rules to Lucid’s C-like syntax presents no theoretical challenge.

The system is defined with respect to some predefined, ordered set of $n$ global variables $g_0$ through $g_{n-1}$, each of which has an associated base type $T_i$. Base types are simply types which do not reference stages — in this example, the only two base types are Unit and Int. Despite the name, global variables are treated as values in the language, not as variables. They can be thought of exactly like ref cells in OCaml, and this line of thinking inspires much of the syntax used for them in the language.

Effects in this system are called stages, and are used to track which global variables have been used so far. Intuitively, the stage $i$ represents the pipeline stage containing $g_i$. We begin typechecking at stage 0, and increment the stage when global variables are used. We can then ensure that the global variables are used in order by only allowing variable $g_i$ to be used if the current stage is at most $i$.

The types in this system are mostly straightforward, but note that functions now have starting and ending stages as well as input and output types, and we have a `ref` type representing the type of a global variable — in general, the type of $g_i$ is $\text{ref}(T_i, i)$.

Expressions in this language are also straightforward, except that we have two operators on global variables: dereference ($\text{ref}$), which returns the current value of the variable, and update ($\text{ref}$), which updates global variable $e_1$ to hold the value of $e_2$. Both of these operators access the global variable, and hence should never be used out-of-order. Note that these operators do not exist in Lucid; instead, there are several built-in functions for performing these accesses.

A.0.1 The Typing Judgement. Our typing judgement has the form $\Gamma, e_1 : \tau, e_2 : \epsilon$, where $\Gamma$ is an environment which maps local variables to values. In English, this judgement can be read as “starting with environment $\Gamma$ at stage $e_1$, the expression $e$ has type $\tau$ and will finish evaluation in stage $e_2$”. The typing rules are presenting in 16.

The most interesting rules here are the `DerEF` and `UP-DATE` rules, as they are the ones which interact with stages. Each first typechecks its subexpression(s) and expects to receive a global variable $g_i$ (i.e. something with `ref` type) as its first argument. Crucially, neither rule can be applied unless the stage after evaluating the subexpression(s) is at most $i$. If this is satisfied, typechecking finishes in stage $i+1$. There is a similar constraint on the function application rule (APP), which specifies that the current stage when beginning to evaluate a function be at most the function’s starting effect.

A.0.2 Extensions in Practice. For clarity, we only present a minimal system here. In practice, the algorithm we implemented for Lucid programs differs in two ways beyond simple syntactic differences. First, it performs type and stage
inference, rather than simply checking the user’s annotations, using an imperative algorithm analogous to Algorithm J [18].

Our algorithm also allows for polymorphic functions, so that a single function definition can be re-used for different input types or at different starting stages. Although adding polymorphism to the system in figure 16 is straightforward, it becomes more difficult when combined with type inference. For example, a function which takes two global variables as arguments and accesses them in order should work for any two inputs where the first input is less than the second. To express this, we extended function types to contain constraints on polymorphic stages which appear in the type. These constraints have the form \(e \leq e\), and can be automatically inferred and checked by the type system.

B SOUNDNESS OF TYPE SYSTEM

In this section we define an operational semantics for the example language defined in 4.2, and prove the soundness of our type system.

B.1 Operational Semantics

Our small-step operational semantics is defined on states of our program, which are three tuples \((G, n, \epsilon)\). Here, \(G\) is an array of values such that \(G[i]\) is the current value of global variable \(g_i\). We write \(G[i]\) for the value in \(G\) at index \(i\), and \(G[i := v]\) for the array with all entries the same as \(G\), but where index \(i\) has value \(v\) instead. We say that \(G\) is well-typed if \(G[i]\) has type \(T_i\) for all \(i\); that is, if \(\emptyset, \epsilon \vdash G[i] : T_i, \epsilon\) for all \(\epsilon\).

\(n\) is an index into the array indicating the next global variable to be used – global variables with index less than \(n\) are inaccessible. Finally, \(\epsilon\) is the expression we are evaluating.

Note the syntactic convention that the metavariable \(v\) will only be used to represent expressions which are values. We use a standard definition of variable substitution, where \(e[v/x]\) means “\(e\) with the value \(v\) substituted for the variable \(x\) wherever it appears”.

B.2 Important Lemmas

We first prove a number of useful lemmas. The Canonical Forms lemma says that typing a value does not change the stage, and that only integer values have type integer, and similarly for other types. The Substitution lemma states that uniformly replacing a variable with a value of the same type does not affect our ability to type an expression. Finally, the weakening lemma says that if an expression typechecks starting at some stage, then it also typechecks starting from any earlier stage.

Lemma (Canonical forms): If \(\emptyset, \epsilon \vdash v : \tau, \epsilon'\), then \(\epsilon = \epsilon'\) and:

- if \(\tau = \text{Int}\) then \(v \in \mathbb{Z}\)

\begin{align*}
\text{PLUS-1} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, e_1 + e_2) \rightarrow (G', n', e'_1 + e_2)
\end{align*}

\begin{align*}
\text{PLUS-2} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, v + e_2) \rightarrow (G', n', v + e'_2)
\end{align*}

\begin{align*}
\text{DEREF-1} & \quad (G, n, e) \rightarrow (G', n', e') \\
& \quad (G, n, \text{!}e) \rightarrow (G', n', \text{!}e')
\end{align*}

\begin{align*}
\text{DEREF-2} & \quad n \leq i \\
& \quad (G, n, \text{!}g_i) \rightarrow (G, i + 1, G[i])
\end{align*}

\begin{align*}
\text{APP-1} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, e_1 \rightarrow e_2) \rightarrow (G', n', e'_1 \rightarrow e'_2)
\end{align*}

\begin{align*}
\text{APP-2} & \quad (G, n, v \rightarrow e) \rightarrow (G', n', v \rightarrow e'_2) \\
& \quad (G, n, v e_2) \rightarrow (G', n', v e'_2)
\end{align*}

\begin{align*}
\text{APP-3} & \quad \epsilon_1 = \text{fun} (x : \tau, \epsilon) \rightarrow e \\
& \quad (G, n, v_1 v_2) \rightarrow (G, n, e[v_2/x])
\end{align*}

\begin{align*}
\text{PLUS-1} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, e_1 + e_2) \rightarrow (G', n', e'_1 + e_2)
\end{align*}

\begin{align*}
\text{PLUS-2} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, v + e_2) \rightarrow (G', n', v + e'_2)
\end{align*}

\begin{align*}
\text{DEREF-1} & \quad (G, n, e) \rightarrow (G', n', e') \\
& \quad (G, n, \text{!}e) \rightarrow (G', n', \text{!}e')
\end{align*}

\begin{align*}
\text{DEREF-2} & \quad n \leq i \\
& \quad (G, n, \text{!}g_i) \rightarrow (G, i + 1, G[i])
\end{align*}

\begin{align*}
\text{APP-1} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, e_1 \rightarrow e_2) \rightarrow (G', n', e'_1 \rightarrow e'_2)
\end{align*}

\begin{align*}
\text{APP-2} & \quad (G, n, v \rightarrow e) \rightarrow (G', n', v \rightarrow e'_2) \\
& \quad (G, n, v e_2) \rightarrow (G', n', v e'_2)
\end{align*}

\begin{align*}
\text{APP-3} & \quad \epsilon_1 = \text{fun} (x : \tau, \epsilon) \rightarrow e \\
& \quad (G, n, v_1 v_2) \rightarrow (G, n, e[v_2/x])
\end{align*}

\begin{figure}[h]
\centering
\begin{align*}
\text{PLUS-1} & \quad (G, n, e_1) \rightarrow (G', n', e'_1) \\
& \quad (G, n, e_1 + e_2) \rightarrow (G', n', e'_1 + e_2)
\end{align*}
\caption{Operational Semantics}
\end{figure}

- if \(\tau = \text{ref}(T, k)\), then \(v = g_k\) and \(T = T_k\).

- if \(\tau = (\text{ref}(T, \epsilon_1)) \rightarrow (\epsilon_{\text{out}}, \epsilon_{\text{out}})\) then \(v = \text{fun} (x : \tau_{\text{in}}, \epsilon_{\text{in}}) \rightarrow e\) and \(\emptyset[x := \tau_{\text{in}}], \epsilon_{\text{in}} \vdash e : \epsilon_{\text{out}}, \epsilon_{\text{out}}\).

Proof: Inversion of the typing relation. ■

Definition: If \(\Gamma\) is a map, let \(\Gamma \setminus x\) denote the same map without a binding for \(x\).

Lemma (Substitution Lemma): If \(\Gamma[x] = \tau\) and \(\emptyset, i \vdash v : \tau, i\) and \(\emptyset, \epsilon \vdash e : \tau', \epsilon'\), then \(\Gamma[x, e : e[v/x]] = \tau', \epsilon'\)

Proof: This is a standard lemma and may be proved for our language in the standard way. ■

Lemma (Weakening): If \(\emptyset, \epsilon \vdash e : \tau, \epsilon'\), and \(\epsilon_1 \leq \epsilon'\), then there is some \(\epsilon'_1 \leq \epsilon'_2\) such that \(\emptyset, \epsilon_1 \vdash e : \tau, \epsilon'_1\).
We prove our soundness theorem in the standard way: by combining progress and preservation lemmas.

**Theorem (Progress):** If \( \emptyset, i \vdash e : \tau, j \) then either \( e \) is a value or for all well-typed \( G \) there exist some \( G', j', e' \) such that \( (G, i, e) \rightarrow (G', j', e') \).

Proof: Structural induction on the typing derivation.

**B.3 Progress**

We prove our soundness theorem in the standard way: by combining progress and preservation lemmas.

**Theorem (Progress):** If \( \emptyset, i \vdash e : \tau, j \) then either \( e \) is a value or for all well-typed \( G \) there exist some \( G', j', e' \) such that \( (G, i, e) \rightarrow (G', j', e') \).

Proof: Structural induction on the typing derivation.

Case INT/UNIT/GLOBAL VARIABLE: This case cannot occur, because values do not evaluate to anything.

Case LOCAL VARIABLE: This case also cannot occur, because the typing judgement contains an empty environment.

Case PLUS: In this case \( e = e_1 + e_2 \) and \( \tau = \text{Int} \). Thus the proof that \( e \) steps must have used either PLUS-1, PLUS-2, or PLUS-3. We also have the premises of the PLUS rule: \( \emptyset, i \vdash e_1 : \text{Int}, k \) and \( \emptyset, k + e_2 : \text{Int}, j \).

- If we used PLUS-1, then we know that \( (G, i, e_1) \rightarrow (G', i', e'_1) \) and \( e' = e'_1 + e_2 \). Thus by induction, \( G' \) is well-typed and \( \emptyset, i' \vdash e'_1 : \text{Int}, k' \) for some \( k' \leq k \). We may use weakening on the second premise to obtain \( \emptyset, k' + e_2 : \text{Int}, j' \) for some \( j' \leq j \), and combine these judgements to show that \( \emptyset, i' + e'_1 + e_2 : \text{Int}, j' \) as required.

- If we used PLUS-2, then we know that \( (G, i, e_2) \rightarrow (G', i', e'_2) \) and \( e' = e_1 + e'_2 \). We also know that \( e_1 \) is a value, so by canonical forms \( e_1 \in \mathbb{Z} \) and \( k = i \). Thus we may combine this with the second premise and use induction to conclude that \( G' \) is well-typed, and that \( \emptyset, i' + e'_2 : \text{Int}, j' \) for some \( j' \leq j \). Since \( e_1 \in \mathbb{Z} \) we may use the INT rule to show that \( \emptyset, i' \vdash e_1 : \text{Int}, j \), and combine this with the previous judgement to show that \( \emptyset, i' + e_1 + e'_2 : \text{Int}, j' \) as required.

- If we used PLUS-3, we know that \( G' = G \) and hence well-typed, \( j = i \), and both \( e_1 \) and \( e_2 \) are values, so \( e_1, e_2 \in \mathbb{Z} \) and thus \( e' \in \mathbb{Z} \) as well. Thus we may simply use the INT rule to show that \( \emptyset, i \vdash e' : \text{Int}, j \) as required.

Case LET: In this case, \( e = \text{let } x = e_1 \text{ in } e_2 \). By induction, either \( e_1 \) is a value or it steps to something. In the latter case we may apply rule LET-1; otherwise, we may apply rule LET-2.

Case DEREF: In this case, \( e = \text{deref } e_1 \). By induction, either \( e_1 \) is a value or it steps to something. In the latter case we may apply rule DEREF-1; otherwise, we have the premise \( \emptyset, i \vdash e_1 : \text{ref}(T, k), j' \) where \( j' \leq k \). Since \( e_1 \) is a value, our canonical forms lemma tells us that \( e_1 = g_k \) and \( j' = i \). Hence \( i = j' \leq k \), so we can apply rule DEREF-2 to show that \( (G, i, e_1) \rightarrow (G, k + 1, G[k]) \).

Case UPDATE: In this case, \( e = e_2 := e_1 \). As in previous parts, the only interesting case is when \( e_1 \) and \( e_2 \) are both values. In that case, as in the DEREF rule, canonical forms tells us that \( e_2 = g_k \) for some \( k \geq i \), and thus we can apply rule UPDATE-3.

Case APP: In this case, \( e = \text{let } x = e_1 \text{ in } e_2 \). The reasoning is analogous to the PLUS case.

**B.4 Preservation**

**Theorem (Preservation):** If \( \emptyset, i \vdash e : \tau, j \), \( G \) is well-typed, and \( (G, i, e) \rightarrow (G', i', e') \), then \( G' \) is also well-typed, and there is some \( j' \leq j \) such that \( \emptyset, i' \vdash e' : \tau, j' \).

Proof: Structural induction on the typing derivation.
either UPDATE-1, UPDATE-2, or UPDATE-3 to show that e steps, and the first two cases are analogous to PLUS-1 and PLUS-2, respectively.

In the UPDATE-3 case, we know that the output value is (), which can trivially be typed using the UNIT rule. So we need only show that $G' = G[k_2 := e_1]$ is well-typed. But since $e_1$ is a value, we must have used the INT or UNIT rule to prove the first premise, and that rule works for all $e$. Thus $G' [k_2]$ has the right type, and all other entries are unchanged, so $G'$ is well-typed.

Case APP: In this case, $e = e_1 e_2$, and we have the premises $∅, i ⊢ e_1 : (τ_{in}, τ_{in}) → (τ_{out}, τ_{out}), k$ and $∅, k ⊢ e_2 : τ_{in}, k_2$ where $k_2 ≤ τ_{in}, τ = τ_{out}$ and $j = τ_{out}$. As usual, we must have used either the APP-1, APP-2, or APP-3 rules here, and the first two cases are again analogous to PLUS-1 and PLUS-2.

If we used the APP-3 rule, then we know that $G' = G$ is well-typed, that $i' = i$, that $e_1 = \text{fun}(x : τ_1, e_1) → e_{body}$ and $e_2$ are both values, and that $e' = e_{body}[e_2/x]$. Since both $e_1$ and $e_2$ are values, by canonical forms we know that $i = k = k_2$.

By the canonical forms lemma on $e_1$, we know that $∅ [x := τ_{in}], e_{in} ⊢ e_{body} : τ_{out}, e_{out}$. Now, $e_2$ is a value, and by the second premise it has type $τ_{in}$; thus by the substitution lemma $∅, e_{in} ⊢ e_{body}[e_2/x] : τ_{out}, e_{out}$. Since $i = k_2 ≤ τ_{in}$, by weakening there is some $e'_{out}$ such that $∅, i ⊢ e_{body}[e_2/x] : τ_{out}, e'_{out}$.

### B.5 Soundness

Finally, we combine the progress and preservation theorems to prove that expressions which typecheck always evaluate - that is, “Well-typed programs do not get stuck”. We denote the transitive closure of the evaluation relation by $→^∗$.

**Theorem (Soundness):** If $∅, e_1 ⊢ e : τ, e_2$, then either $e$ is a value, or $e → e'$ and there is some $e'_1$ such that $∅, e'_1 ⊢ e' : τ, e_2$.

Proof: Inductively apply preservation to show that $G_e$ and $e_2$ are well-typed, then apply progress to show that $e_2$ is either a value, or another step can be taken.

### C LUCID GRAMMAR

```
declarations ::= 'const' ⟨type⟩ ⟨const_var_id⟩ '=' ⟨value⟩ · · · | 'global' ⟨var_id⟩ =>'new' ⟨module_id⟩ '<(size)>'⟨⟨(args)⟩⟩ · · · | event_qualifier 'event' ⟨event_id⟩ '(' ⟨params⟩ '⟨⟨(stmt)⟩⟩' · · · | 'handle' ⟨event_id⟩ '(' ⟨params⟩ '⟨⟨(stmt)⟩⟩' · · · | 'fun' ⟨type⟩ ⟨function_id⟩ '(' ⟨params⟩ '⟨⟨(stmt)⟩⟩' · · · | (memop_decl) (function operating on persistent state)

statements ::= ⟨no_op⟩ · · · | ⟨stmt⟩ · · · | 'if' '(' ⟨expr⟩ ')' '(' ⟨stmt⟩ ')' 'else' '(' ⟨stmt⟩ ')' · · · | (type) ⟨var_id⟩ '[' ⟨value⟩ ']' · · · | 'local_var' ⟨var_id⟩ '=' ⟨value⟩ · · · | 'call' ⟨value⟩ · · · | 'return' ⟨expr⟩ · · · | 'generate' ⟨expr⟩ · · · | 'produce' ⟨expr⟩ · · · | 'call' ⟨value⟩ · · · | 'return' ⟨value⟩ · · ·

expressions ::= ⟨value⟩ · · · | ⟨expr⟩ '(' ⟨bin_op⟩ ⟨expr⟩ ')' · · · | ⟨callable_id⟩ '(' ⟨⟨(args)⟩⟩ · · · | 'hash' '<size>' '(' ⟨⟨(args)⟩⟩ · · ·

miscellaneous ::= ⟨value⟩ · · · | ⟨bin_op⟩ · · · | ⟨params⟩ · · · | ⟨⟨(expr)⟩⟩ · · · | ⟨⟨(stmt)⟩⟩ · · · | ⟨⟨(call)⟩⟩ · · · | ⟨⟨(module_id)⟩⟩ · · · | ⟨⟨(event)⟩⟩ · · ·
```

Figure 18: The Lucid language grammar.

```
memop declarations ::= 'memop' ⟨callable_id⟩ '(' ⟨⟨m_params⟩⟩ ')' '(' ⟨⟨m_stmt⟩⟩ ')' · · · | ⟨⟨m_stmt⟩⟩ ::= ⟨⟨stmt⟩⟩ · · · | ⟨⟨m_param⟩⟩ ::= (⟨⟨stmt⟩⟩ · · ·)

memop return statements ::= ⟨⟨stmt⟩⟩ ::= 'return' ⟨⟨value⟩⟩ · · · | ⟨⟨call⟩⟩ ::= (⟨⟨value⟩⟩ · · · | ⟨⟨return⟩⟩ ::= (⟨⟨stmt⟩⟩ · · · | ⟨⟨test⟩⟩ ::= (⟨⟨expr⟩⟩ · · · | ⟨⟨mem⟩⟩ ::= (⟨⟨const⟩⟩ · · ·

Figure 19: The restricted grammar of memory operations. Each memop parameter ⟨⟨m_param⟩⟩ may only appear once in a ⟨⟨return⟩⟩ and ⟨⟨test⟩⟩.

17
D SINGLE-OPERATION P4 TABLES

The back-end of Lucid’s compiler translates primitive Lucid statements generated by the front end into single-operation P4 tables. There are three types of primitive Lucid statements, that each use a different template to translate into a P4 table.

**Binary operations:** A binary operation evaluates a binary expression over local variables and assigns its result to a (possibly new) local variable. The compiler translates a binary operation into a P4 action wrapped with a and table. For example, \( \text{idx} = \text{idx} + \text{NUM_PORTS}; \) translates to:

```p4
action do_idx_eq_0 {idx = idx + NUM_PORTS;}
table idx_eq_0 {
  actions = {do_idx_eq_0;}
  const default_action = {do_idx_eq_0;}
}
```

**Stateful operations:** A stateful operation is a call to a data structure method that operates on a global variable. The compiler translates this call to a table that executes a cogenerated `RegisterAction`. The body of the `RegisterAction` is a direct (but verbose) translation of the `memops` passed in the call. For example, here is the register action generated by the call `Array.fset(tcp_cts, port, plus, 1);`:

```p4
RegisterAction<bit<32>,bit<32>,bit<32>> (tcp_cts) fset_1 = {
  void apply(inout bit<32> mem, out bit<32> ret) {
    mem = mem + 1;
  }
};
```

The translation from `memop` to `RegisterAction` can be direct because the syntactic restrictions of `memops` guarantees that they can compile to the Tofino’s stateful ALUs.

**Branch operations:** A branch operation is an `if/else` statement that evaluates a simplified conditional expression. The simplified expression is a conjunction or disjunction of clauses that each compare a local variable with a constant. A branch operation translates to a table that evaluates the conditional expression using ternary rules and calls a true or false action. For example, `if(proto != TCP)` translates to:

```p4
action if_0_true(); action if_0_false();
table if_0 {
  keys = {proto : ternary;}
  actions = {if_0_true; if_0_false;}
  entries = {
    (TCP) : if_0_false;
    (_,) : if_0_true;
  }
}
```

The compiler also generates a table of this form to select between handlers based on an event ID field.